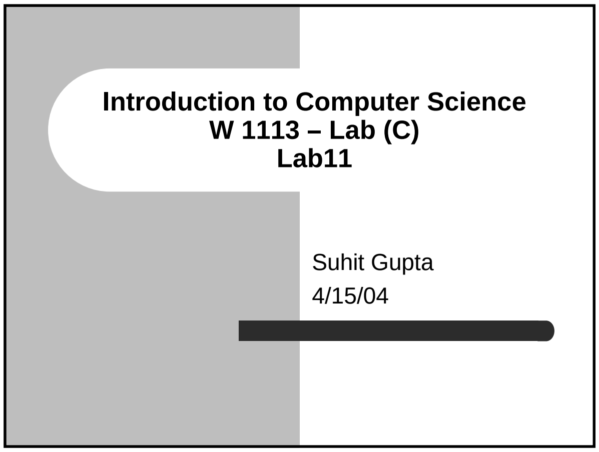
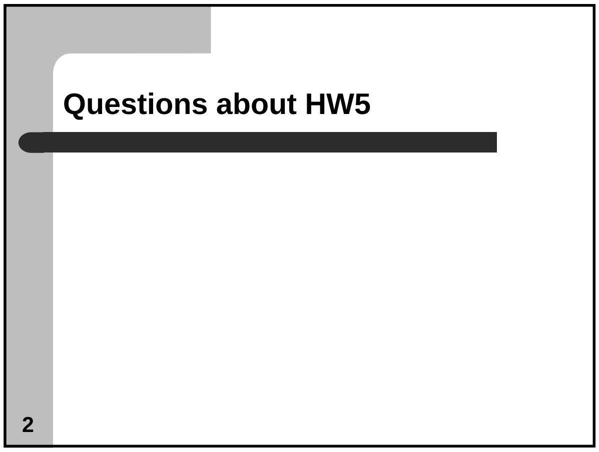
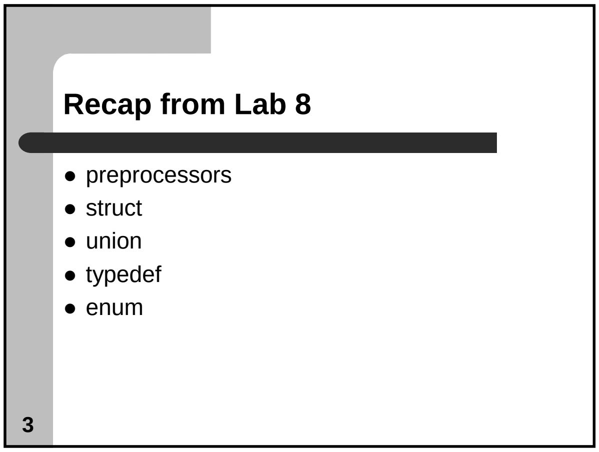


Introduction to Computer Science
W 1113 – Lab (C)
Lab11

Suhit Gupta
4/15/04

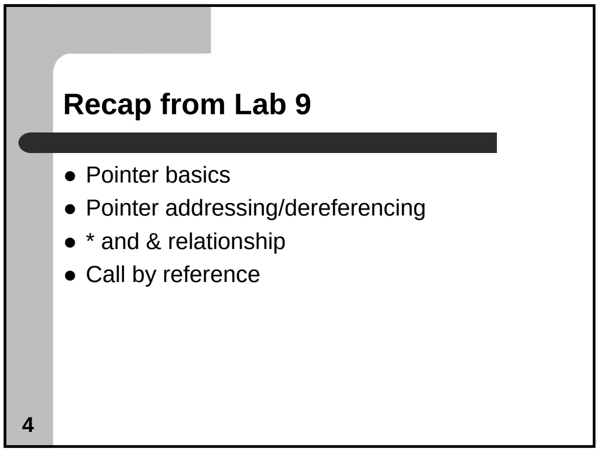


Questions about HW5



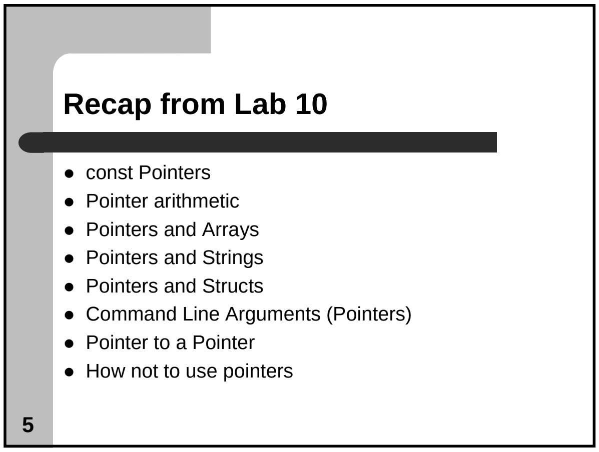
Recap from Lab 8

- preprocessors
- struct
- union
- typedef
- enum



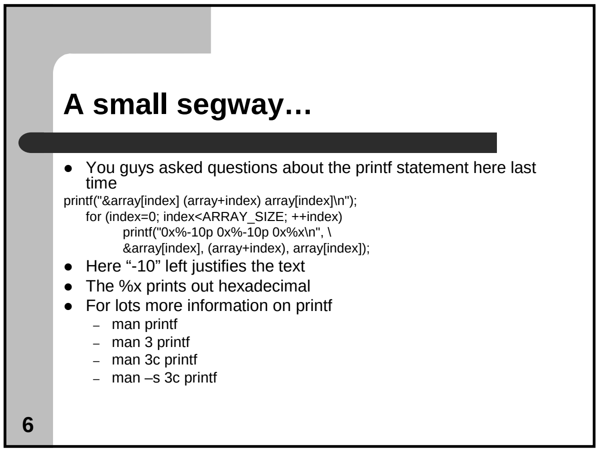
Recap from Lab 9

- Pointer basics
- Pointer addressing/dereferencing
- * and & relationship
- Call by reference



Recap from Lab 10

- const Pointers
- Pointer arithmetic
- Pointers and Arrays
- Pointers and Strings
- Pointers and Structs
- Command Line Arguments (Pointers)
- Pointer to a Pointer
- How not to use pointers



A small segway…

- You guys asked questions about the printf statement here last time

```
printf("&array[index] (array+index) array[index]\n");
    for (index=0; index<ARRAY_SIZE; ++index)
            printf("0x%-10p 0x%-10p 0x%x\n", \
            &array[index], (array+index), array[index]);
```

- Here “-10” left justifies the text
- The %x prints out hexadecimal
- For lots more information on printf
  - man printf
  - man 3 printf
  - man 3c printf
  - man –s 3c printf

Storing an indeterminate amount of data

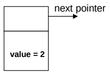
- How would you store an indeterminate amount of data?
- You create a bank, but you don't know how many accounts you are going to have
- Two ways to fix this
  - Growable arrays
    - If the array fills up, create an array twice its size and copy all the elements over
  - Linked Lists

7

Pointers and linked lists

- Instead of statically declaring an array, we can create a bunch of nodes and link them together

```
struct node {
    struct node *next_ptr;
    int value;
}
```



- If you wanted to create a large number of these nodes

```
struct node node_1;
struct node node_2;
```

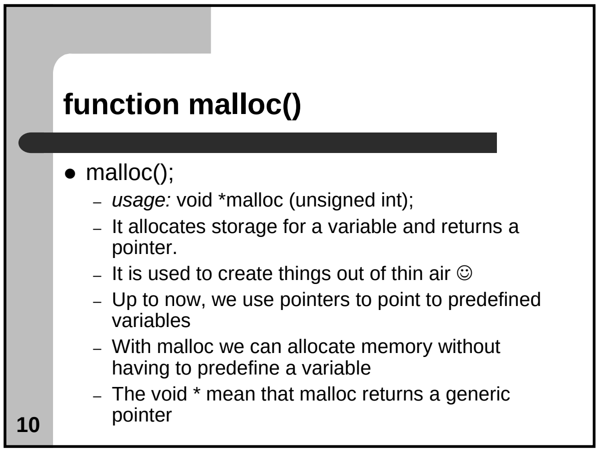
- BTW, do you guys know what linked lists are?

8

Pointers and linked lists II

- However, you can only declare a limited number of nodes.
  - well, ok, so you can create a lot, but if you didn't know how many you would need, then you have a problem.
- Therefore you can allocate memory dynamically

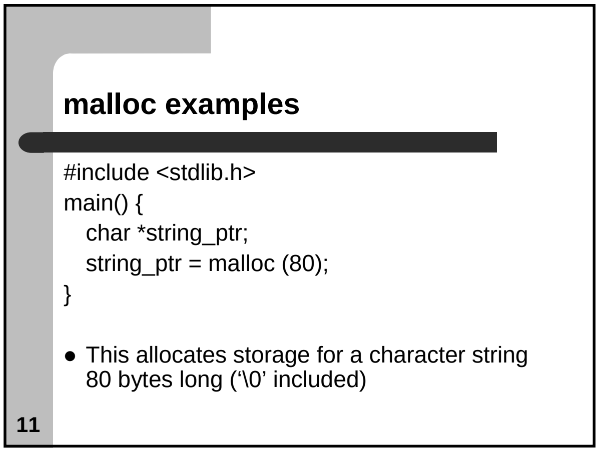
9



function malloc()

- malloc();
  - *usage:* void *malloc (unsigned int);
  - It allocates storage for a variable and returns a pointer.
  - It is used to create things out of thin air ☺
  - Up to now, we use pointers to point to predefined variables
  - With malloc we can allocate memory without having to predefine a variable
  - The void * mean that malloc returns a generic pointer

10

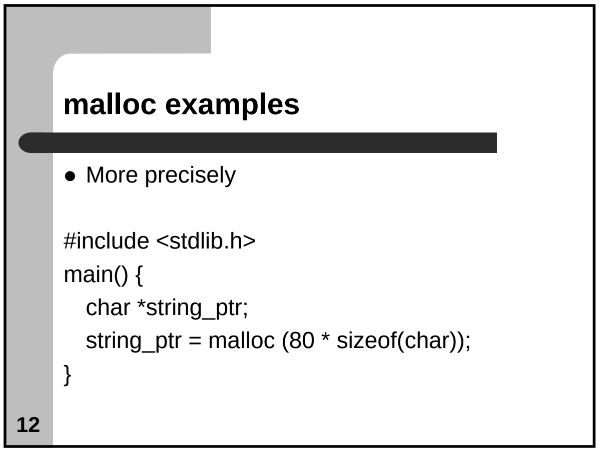


malloc examples

```
#include <stdlib.h>
main() {
   char *string_ptr;
   string_ptr = malloc (80);
}
```

- This allocates storage for a character string 80 bytes long ('\0' included)

11

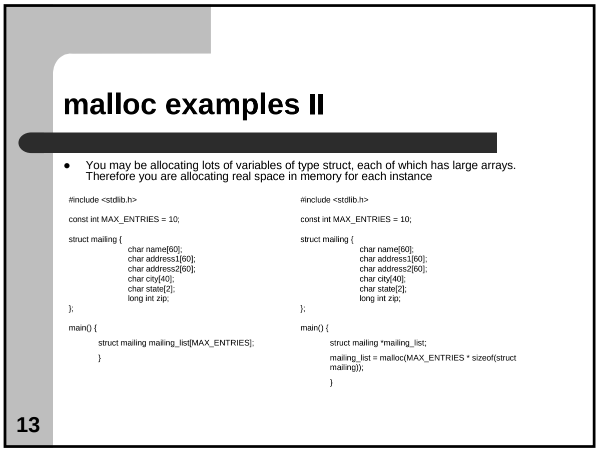


malloc examples

- More precisely

```
#include <stdlib.h>
main() {
   char *string_ptr;
   string_ptr = malloc (80 * sizeof(char));
}
```

12



malloc examples II

- You may be allocating lots of variables of type struct, each of which has large arrays. Therefore you are allocating real space in memory for each instance

```
#include <stdlib.h>

const int MAX_ENTRIES = 10;

struct mailing {
            char name[60];
            char address1[60];
            char address2[60];
            char city[40];
            char state[2];
            long int zip;
};

main() {
      struct mailing mailing_list[MAX_ENTRIES];
      }
```

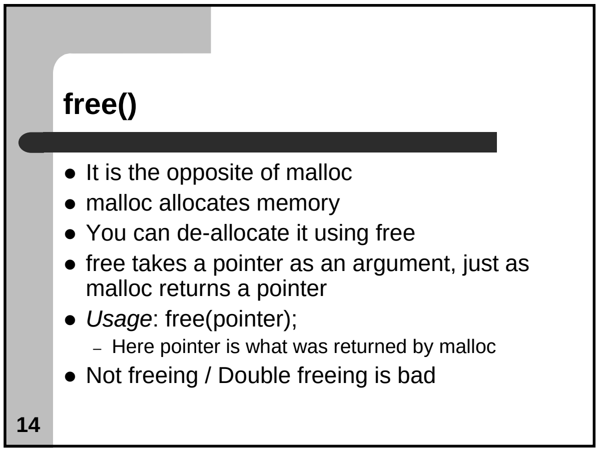
```
#include <stdlib.h>

const int MAX_ENTRIES = 10;

struct mailing {
            char name[60];
            char address1[60];
            char address2[60];
            char city[40];
            char state[2];
            long int zip;
};

main() {
      struct mailing *mailing_list;
      mailing_list = malloc(MAX_ENTRIES * sizeof(struct
      mailing));
      }
```

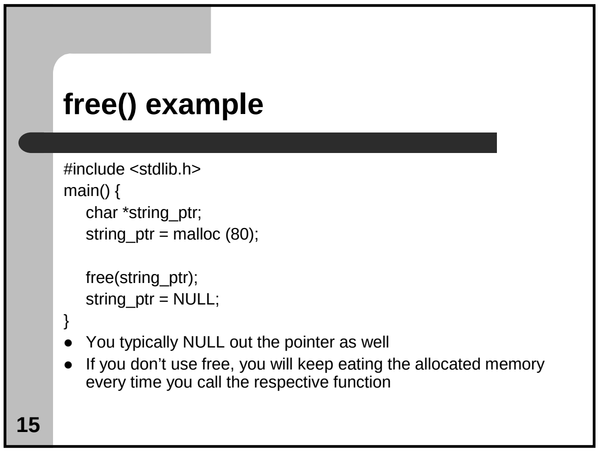
13



free()

- It is the opposite of malloc
- malloc allocates memory
- You can de-allocate it using free
- free takes a pointer as an argument, just as malloc returns a pointer
- *Usage*: free(pointer);
  - Here pointer is what was returned by malloc
- Not freeing / Double freeing is bad

14



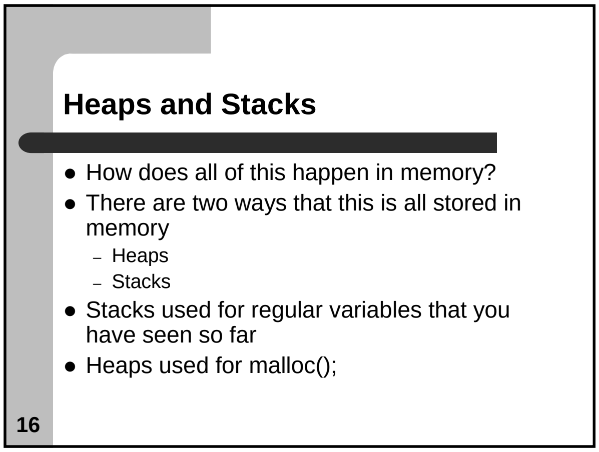
free() example

```
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc (80);

    free(string_ptr);
    string_ptr = NULL;
}
```

- You typically NULL out the pointer as well
- If you don't use free, you will keep eating the allocated memory every time you call the respective function

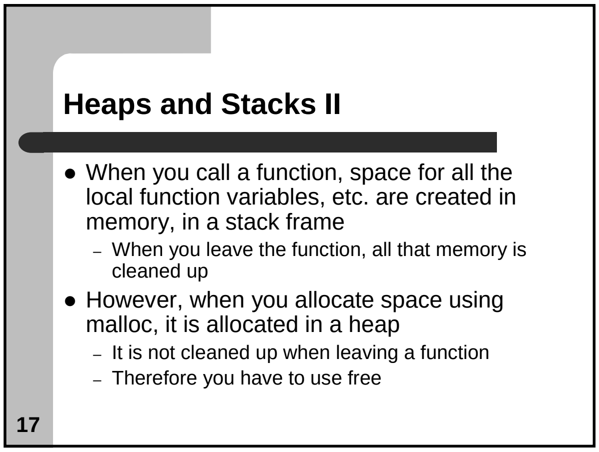
15



Heaps and Stacks

- How does all of this happen in memory?
- There are two ways that this is all stored in memory
  - Heaps
  - Stacks
- Stacks used for regular variables that you have seen so far
- Heaps used for malloc();

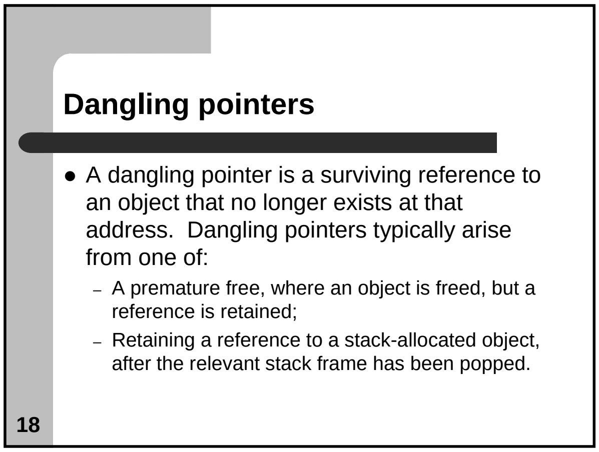
16



Heaps and Stacks II

- When you call a function, space for all the local function variables, etc. are created in memory, in a stack frame
  - When you leave the function, all that memory is cleaned up
- However, when you allocate space using malloc, it is allocated in a heap
  - It is not cleaned up when leaving a function
  - Therefore you have to use free

17



Dangling pointers

- A dangling pointer is a surviving reference to an object that no longer exists at that address. Dangling pointers typically arise from one of:
  - A premature free, where an object is freed, but a reference is retained;
  - Retaining a reference to a stack-allocated object, after the relevant stack frame has been popped.

18

Bad code (preliminary free)

```
int main(void) {
  int *result = malloc(sizeof(int));
  *result = 6;
  free(result);
  printf(“result is %d\n”, *result);
}
```

19

Bad code (stack memory)

```
int main(void) {
  int *result = square(6);
  printf(“result is %d\n”, *result);
}

int *square(int i) {
  int j = i * i;
  return &j;
}
```

20

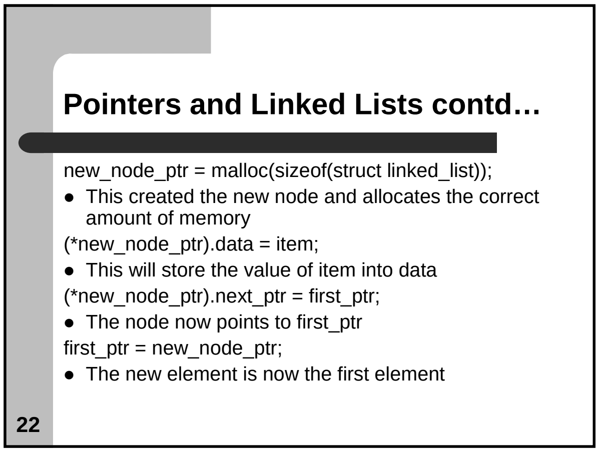
Back to linked lists

- So how does malloc help us here?

```
struct linked_list {
    char data[30];
    struct linked_list *next_ptr;
}
struct linked_list *first_ptr = NULL;
```

- So we want to use malloc instead of creating an array of linked lists that will limit the number of nodes in the linked list to the size of the array
- How can we do this?

21



Pointers and Linked Lists contd…

new_node_ptr = malloc(sizeof(struct linked_list));

- This created the new node and allocates the correct amount of memory

(*new_node_ptr).data = item;

- This will store the value of item into data

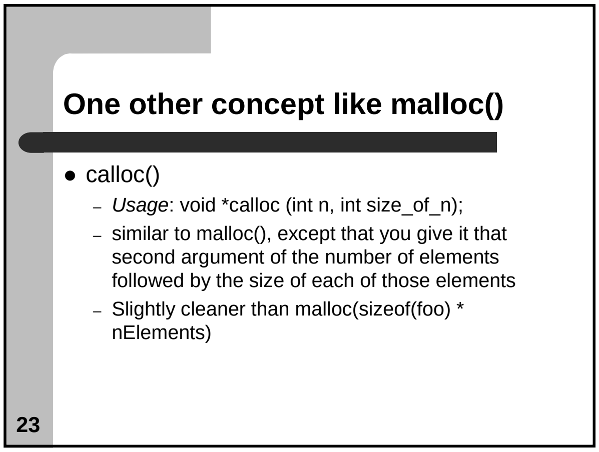
(*new_node_ptr).next_ptr = first_ptr;

- The node now points to first_ptr

first_ptr = new_node_ptr;

- The new element is now the first element

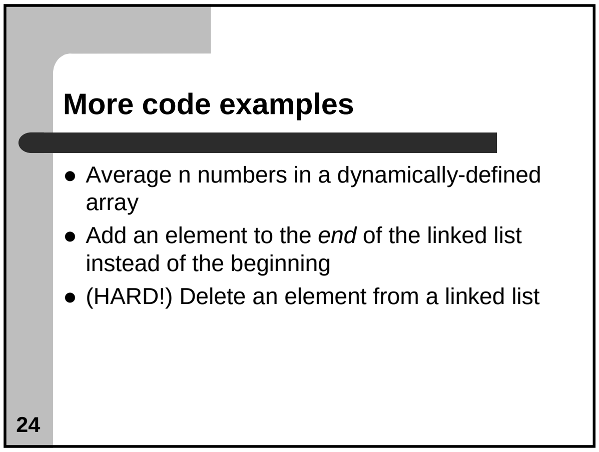
22



One other concept like malloc()

- calloc()
  - *Usage*: void *calloc (int n, int size_of_n);
  - similar to malloc(), except that you give it that second argument of the number of elements followed by the size of each of those elements
  - Slightly cleaner than malloc(sizeof(foo) * nElements)

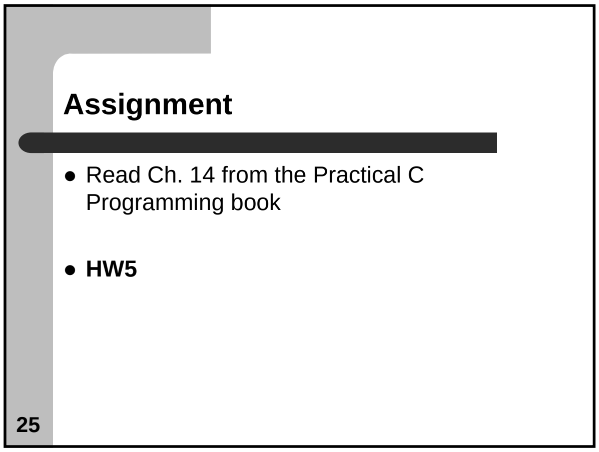
23



More code examples

- Average n numbers in a dynamically-defined array
- Add an element to the *end* of the linked list instead of the beginning
- (HARD!) Delete an element from a linked list

24



Assignment

- Read Ch. 14 from the Practical C Programming book
- **HW5**

25